# Textual Analysis and Software Quality: Challenges and Opportunities

Gabriele Bavota[1], Andrea De Lucia[2], Rocco Oliveto[3],
Fabio Palomba[2], Annibale Panichella[2]
[1]*Department of Engineering, University of Sannio*
*Palazzo ex Poste, Via Traiano, 82100 Benevento, Italy*
*gbavota@unisannio.it*
[2]*Department of Management and Information Technology, University of Salerno*
*Via Giovanni Paolo II, 84084 Fisciano, Italy*
*adelucia@unisa.it, fabio.palomba.89@gmail.com, apanichella@unisa.it*
[3]*Department of Bioscience and Territory, University of Molise*
*c.da Fonte Lappone, 86090 Pesche (IS), Italy*
*rocco.oliveto@unimol.it*

***Abstract.*** *Source code lexicon (identifier names and comments) has been used – as an alternative or as a complement to source code structure – to perform various kinds of analyses (e.g., traceability recovery). All these successful applications increased in the recent years the interest in using textual analysis for improving and assessing the quality of a software system. In particular, textual analysis could be used to identify refactoring opportunities or ambiguous identifiers that may increase the program comprehension burden by creating a mismatch between the developers' cognitive model and the intended meaning of the term, thus ultimately increasing the risk of fault proneness. In addition, when used "on-line" during software development, textual analysis could guide the programmers to select better identifiers aiming at improving the quality of the source code lexicon. In this paper, we overview research in text analysis for the assessment and the improvement of software quality and discuss our achievements to date, the challenges, and the opportunities for the future.*

**Keywords:** Software quality, Textual analysis, Survey.

## 1. Introduction

During software development and evolution a variety of software artifacts are created, such as, requirements, bug descriptions, documentation, source code, test cases, etc. These artifacts have different representations and contain different types of information, i.e., structural (e.g., control and data flow), dynamic (e.g., execution traces), process (e.g., CVS logs), and textual (e.g., identifiers and comments in source code, documentation). The textual

information captures knowledge about the problem and solution domain, about developer's intentions, client demands, etc. and it is the most common type of information present in software. Text is also the common form to represent information among various artifacts at different abstraction levels. Among other things, developers use textual information to understand what a specific piece of code implements and make decisions during their daily tasks. For very small software systems, developers could read all the text found in software artifacts and extract and use only the information that is useful for their current task. However, as the size and complexity of the system increases, tools are required to extract, analyze, and retrieve this information to the developers.

For these reasons, in recent and past years, textual analysis (TA) has been successfully applied to leverage the textual information and help developers in several software engineering tasks, such as traceability recovery [Antoniol et al, 2002], impact analysis [Canfora and Cerulo, 2005], clone detection [Marcus and Maletic, 2001], feature location [Poshyvanyk et al, 2007]. The use of TA in software engineering has demonstrated to be effective for various reasons:

- it is *lightweight* and to some extent *independent on the programming language*, as it does not require a full source code parsing, but only its tokenization and (for some applications) lexical analysis;
- it *provides information complementary to what structural or dynamic analysis can provide* [Marcus et al, 2008];
- it models software artifacts as textual documents, thus *it can be applied to different kinds of artifacts* (i.e., it is not limited to the source code) and, above all, can be used to perform combined analysis of different kinds of artifacts (e.g., requirements and source code), as in the case of traceability recovery.

All these successful applications increased in the recent years the interest in using TA for improving and assessing the quality of software systems. This paper offers an overview of the process that is usually followed when using TA techniques to support software engineering tasks, focusing the attention on activities aimed at improving software quality. Other than discussing the state of the art, the paper also presents challenges and opportunities for the future. Thus, it represents a useful roadmap for both practitioners, who want to know how to use TA in their working environment, and researchers, who want to get closer and doing research on this topic.

**Paper structure**. Section 2 provides background information on how to extract and manage textual information. Sections 3 and 4 describe how textual information can be used to measure quality aspects, i.e., cohesion and coupling, and identifying refactoring opportunities, respectively. Finally, Section 6 concludes the paper highlighting challenges and new horizons.

## 2. Background

TA has been proposed for a variety of software engineering tasks, and making use of different text retrieval (TR) techniques. No matter the particular task or

retrieval technique used, an approach based on TR generally follows the same process:

- *extracting text documents from software artifacts (the corpus)*;
- *indexing the corpus*;
- *computing similarity between documents*.

Each step is explained in detail in the following subsections.

## 2.1 Extracting the corpus

The first step in using TR techniques is to define a collection of text documents, also known as *corpus*, which are extracted from the software artifacts. Documents can be extracted at different granularities from an artifact. For example, in the case of source code, a document could be represented by structural elements of the code, such as a class. In the case of textual software documentation, sentences, paragraphs, sections, or chapters could represent the documents. Thus, a software artifact may be represented by one or more documents in the corpus. The document granularity needs to be decided up front according to the needs of the task at hand and can influence greatly the results of text retrieval.

Once the corpus is extracted, a few optional, corpus normalization steps can be performed before the documents are indexed by the text retrieval technique [Baeza-Yates and Ribeiro-Neto, 1999]:

- *term extraction,* aimed at extracting words from the artifacts and removing anything useless (e.g., punctuation or programming language operators);
- *identifier splitting,* aimed at splitting composite identifiers. This step is important to align source code and documentation vocabulary, since identifiers are often composed of several concatenated dictionary words. The simplest approaches for identifier splitting are based on common conventions for separating words in identifiers, such as using camel case, underscore, numbers and symbols as separators. For example, `SETpointer`, `set_pointer`, `setPointer` would be all split to *set* and *pointer*. More advanced techniques make use of dictionaries and abbreviation lists to identify words in the cases where common naming conventions are not used, e.g., the identifier `setptr` would be split into *set* and *pointer* based on these techniques [Guerrouj et al, 2012];
- *term filtering,* aimed at removing common terms, referred to as “stop words” (e.g., articles, prepositions, common use verbs, or programming language keywords). Words shorter than a given length (e.g., shorter than three characters) are removed as well.

Morphological analysis of the extracted words is often performed to bring back words to the same root (e.g., by removing plurals to nouns, or verb conjugations). The simplest way to do morphological analysis is by using a stemmer, e.g., the Porter stemmer [Porter, 1980]. Other stemmers used by

researchers in software engineering are WordNet's morphstr function[1] and the Snowball stemmer[2].

## 2.2 Indexing the corpus

The extracted information is stored in a *m x n* matrix (called *term-by-document matrix*), where *m* is the number of terms occurring in all artifacts, and *n* is the total number of artifacts in the repository. A generic entry $w_{i,j}$ of this matrix denotes a measure of the weight (i.e., relevance) of the $i^{th}$ term in the $j^{th}$ document [Baeza-Yates and Ribeiro-Neto, 1999]. Such weight, independently by the used technique, is based on two criteria: how well they describe the current document (*local weight*) and how they relate to the entire corpus (*global weight*). A widely used measure is the *tf-idf* (*term frequency-inverse document frequency*), which gives more importance to words having a high frequency in a document (*tf*) and appearing in a small number of documents, thus having a high discriminating power (high *idf*).

## 2.3 Computing similarity between documents

Based on the *term-by-document matrix* representation, different Information Retrieval (IR) methods can be used to compute similarity between documents aiming at deriving latent patterns between them. A survey of available research papers reveals that probabilistic models, Vector Space Model (VSM), its extension Latent Semantic Indexing (LSI), and Latent Dirichlet Allocation (LDA) are the four most frequently used IR methods in software engineering. In the following we describe in details VSM and LSI since these two techniques have been used to assess software quality. However, the interested reader can find more details on IR methods the book by Baeza-Yates and Ribeiro-Neto [1999].

In the VSM, a document is represented by a vector of terms, i.e., column of the *term-by-document matrix*. Since any document contains a limited set of terms, while the vocabulary (all the terms in the documents) can be millions of terms, most document vectors are very sparse and they generally operate in a positive quadrant of the vector space, i.e., no term is assigned a negative value. In VSM the angle between two vectors is used as a measure of divergence between the vectors, and the cosine of the angle is used as the numeric similarity between the corresponding documents. The cosine has a property indicating 1.0 for identical vectors (very similar documents) and 0.0 for orthogonal vectors (completely different document). A common criticism of VSM is that it does not take into account relations between terms [Deerwester et al, 1990], e.g., having “automobile” in one document and “car” in another document does not contribute to the similarity measure between these two documents.

LSI [Deerwester et al, 1990] was developed to overcome the synonymy and polysemy problems, which occur with the VSM model. In LSI the dependencies between terms and documents, in addition to the associations between terms and documents, are explicitly taken into account. LSI assumes that there is an

[1] http://wordnet.princeton.edu/
[2] http://snowball.tartarus.org/

underlying or “latent structure” in word usage that is partially obscured by variability in word choice, and uses statistical techniques to estimate this latent structure. For example, both “car” and “automobile” are likely to co-occur in different documents with related terms, such as “motor”, “wheel”, etc. LSI exploits information about co-occurrence of terms (i.e., latent structure) to automatically discover synonymy between different terms.

Specifically, LSI defines a term-by-document matrix as well as VSM. Then it applies the Singular Value Decomposition (SVD) [Deerwester et al, 1990] to project the original term-by-document matrix into a reduced space of concepts. The size of this space is *k*, that is much lower than *n*, i.e., number of terms. The cosine of the angle between two vectors in the *k*-space represents the similarity of the two documents (terms, respectively) with respect to the concepts they share. In this way, SVD captures the underlying structure in the association of terms and documents. Terms that occur in similar documents, for example, will be near each other in the space of concepts, even if they never co-occur in the same document. This also means that some documents that do not share any word, but share similar words may nonetheless be near in the space of concepts. The choice of *k* is critical and the proper way to make such a choice is an open issue in the factor analysis literature [Deerwester et al, 1990].

The obtained similarity measure is used to support different software engineering tasks. For instance, given a software artifact used as query (e.g., requirement), the similarity measure can be used to find similar source code classes in order to identify traceability links between requirements and source code [Antoniol et al, 2002] or to perform impact analysis [Canfora and Cerulo, 2005]. In the next sections we show how textual similarity can be used to monitor and improve the internal quality of software systems.

## 3.Capturing cohesion and coupling

Cohesion is a desirable property of software as it positively impacts understanding, reuse, and maintenance. There are several metrics to measure the cohesion of a software component. These metrics are generally based on structural information extracted from the source code, such as attribute references [Chidamber and Kemerer, 1998]. However, information contained in identifiers and comments could be worthwhile to measure the cohesiveness of software component. Specifically, two components are conceptually related if their (domain) semantics are similar, i.e., they perform conceptually similar actions. To this aim, Marcus *et al.* [2008] proposed a semantic measure, called Conceptual Cohesion of Classes (C3), to capture the cohesion of a class inspired by the mechanisms used to measure textual coherence in cognitive psychology and computational linguistics. In order to capture the semantic cohesion of a class, LSI is used to represent each method as a real-valued vector that spans a space defined by the vocabulary extracted from the code. The conceptual similarity between two methods (CSM) is then calculated as the cosine of the angle between their corresponding vectors. Thus, the higher the value of CSM the higher the similarity between two methods. The average value between the CSM of all passable pairs of methods of a class represents the

Conceptual Cohesion of the class. In short, the measure captures relationships between the comments, identifiers, and other text present in the methods, based on word usages in the entire code. It is clear that C3 depends on the consistency of naming conventions used in the source code as well as on the comments contained in it. An empirical study conducted on three open-source systems has been performed to compare the novel metric with a set of existing metrics based on structural information, e.g., Lack of COhesion in Methods (LCOM) [Chidamber and Kemerer, 1998] and Coh [Briand et al, 1998]. The results achieved indicated that C3 captures different aspects of class cohesion compared to any of the existing cohesion measures.

Following the same idea, Poshyvanyk *et al.* [2009] proposed the Conceptual Coupling Between Classes (CCBC). Specifically, they used LSI to measure the similarity between methods of two classes. The average similarity of all possible pairs of the two classes provides an indication of the conceptual coupling between the two classes. The higher the similarity, the higher the coupling between them, i.e., the methods of the two classes perform conceptually similar actions. An empirical study indicated that CCBC provides orthogonal information as compared to coupling metrics based on structural information, e.g., Coupling between Objects (CBO) [Chidamber and Kemerer, 1998].

## 3.1 Improving defect prediction

Empirical studies have indicated that semantic metrics capture different aspects of cohesion and coupling as compared to structural metrics. This suggests that the combination of semantic and structural metrics is a more complete cohesion indicator than any combination of structural metrics. In order to verify such a conjecture, Marcus *et al.* [2008] used the C3 metric to improve defect prediction models[3] based only on structural cohesion metrics. They performed a study where logistic regression models were used to predict the defects of classes of version 1.6 and 1.7 of Mozilla. The considered predictors were the metrics from the CK suite [Chidamber and Kemerer, 1998] and the C3. Results indicated that the three models with the best precision in prediction defect-prone classes are C3+LCOM3, C3+LCOM1, and C3+Coh. Thus, C3 is a useful indicator of an external property of classes in OO systems, that is, the defect proneness of classes. More importantly, the results support the initial conjecture that the combination of C3 with other cohesion metrics allows to build superior models for detecting defect prone classes.

## 3.2 Capturing developer perception of quality

From the analysis of the literature we can derive that semantic metrics complement structural metrics and are useful to measure the quality of software components. However, little is known about how developers actually perceive quality and if existing measures align with this perception.

[3] One of the most interesting applications of cohesion metrics in software engineering is to predict defects in classes.

In order to bridge this gap, Bavota *et al.* [2013a] empirically investigated how a specific quality attribute, namely class coupling – as captured by structural, dynamic, semantic, and logical coupling measures – aligns with developers' perception of coupling. The study has been conducted on three Java open-source systems, i.e., ArgoUML, JHotDraw and jEdit, and involved 64 students, academics, and industrial practitioners from around the world, as well as 12 active developers of these three systems. In the context of the study, the authors asked participants to assess the coupling between pairs of classes exhibiting high (low) coupling as indicated by the four different types of coupling and provide their ratings and some rationale.

The results indicate that the peculiarity of the semantic coupling measure allows it to better estimate the mental model of developers than the other coupling measures. In other words, when the semantic measure indicates high (low) coupling between two classes, also the developers feel the same. This is because, in several cases, the interactions between classes are encapsulated in the source code vocabulary, and cannot be easily derived by only looking at structural relationships, such as method calls.

## 4.Identifying refactoring opportunities

During software evolution the internal structure of the system undergoes continuous modifications. These continuous changes push away the source code from its original design, often reducing its quality, including class cohesion [Fowler, 1999]. In such scenarios a refactoring of the system is recommended. Refactoring has been defined as "*the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure*" [Fowler, 1999].

Typical advantages of refactoring include improved readability and reduced complexity of source code, a more expressive internal architecture and better software extensibility [Fowler, 1999]. For these reasons, refactoring is advocated as a good programming practice to be continuously performed during software development and maintenance. However, despite its advantages, performing refactoring in non-trivial software systems might be very challenging. First, the identification of refactoring opportunities in large systems is very difficult, due to the fact that the design flaws are not always easy to identify. Second, when a design problem has been identified, it is not always easy to apply the correct refactoring operation to solve it. All these observations highlight the need for (semi)automatic approaches supporting the software engineer in (i) identifying refactoring opportunities (i.e., design flaws) and (ii) designing and applying a refactoring solution.

Existing approaches for suggesting refactoring opportunities generally use metrics, e.g. cohesion metrics, which capture structural relationships between the members of a class, e.g., method-to-attribute references. However, semantic metrics have been proved to complement structural cohesion metrics. Based on this observation, Bavota *et al.* [2013c] proposed a method for automating the Extract Class refactoring. The proposed approach analyzes (structural and semantic) relationships between the methods in a class to

identify chains of strongly related methods. The identified method chains are used to define new classes with higher cohesion than the original class, while preserving the overall coupling between the new classes and the classes interacting with the original class. The proposed approach has been empirically evaluated on open source systems in order to assess how good and useful the proposed refactoring solutions are considered by software engineers and how well the proposed refactorings approximate the refactorings done by the original developers. Results indicate that (i) the semantic measure plays a crucial role in identifying meaningful refactorings; and (ii) the proposed solutions are useful in guiding refactorings. Following the same idea, Bavota *et al.* [2013d] also proposed an approach for suggesting Extract Package refactoring aiming at improving cohesion in packages.

Semantic measures were also used to identify Move Class refactoring [Bavota et al, 2013c]. The proposed approach, called R3 (Rational Refactoring via RTM), analyzes underlying latent topics in source code as well as structural dependencies to recommend (and explain) refactoring operations aiming at moving a class to a more suitable package. R3 has been evaluated in two empirical studies. The results of the first study conducted on nine software systems indicate that R3 provides a coupling reduction from 10% to 30% among the software modules. The second study with 62 developers indicates that more than 70% of the recommendations (and explanations) where considered meaningful from a functional point of view.

## 5.Conclusion and directions for future work

Textual analysis has been widely and successfully applied in software engineering for supporting several tasks. However, they also have some weaknesses, and poses challenges for researchers:

- ***$C_1$**: Quality of source code lexicon.* TA depends on the quality of the lexicon: a bad lexicon often means inaccurate – if not completely wrong – results. There are two common problems in the TA of software artifacts. The first is represented by the presence of inconsistent terms in related documents (e.g., requirements express some concepts using certain words, whereas the source code uses synonyms or abbreviations). The second problem is related to the presence of “noise” in software artifacts, for example due to recurring terms that do not bring information relevant for the analysis task, e.g. programming language keywords and terms that are part of a specific document template, such as a test case specification, a use case, or a bug report.
- ***$C_2$**: Setting of textual analysis technique.* TA techniques require configuring different components and their respective parameters, such as type of pre-processors (e.g., splitting identifiers, term weighting schema, stemmers). Despite this overwhelming popularity of IR methods in SE research, most of the proposed approaches are based on ad-hoc methods to configure these solutions, components, and their configurations, thus resulting oftentimes in suboptimal performance of such promising analysis methods. In other

words, existing methods for designing IR-based solutions for SE tasks is currently based on art, rather than science. This also makes the practical use of IR-based processes quite difficult and undermines the technology transfer to software industry.

Clearly, these challenges represent on one hand an obstacle for the technology transfer in industry, on the other hand a fertile ground for the definition of new solutions aimed at advancing the state of the art. We briefly outline some opportunities for future work aimed at mitigating the weaknesses of TA for software engineering:

- ***$O_1$**: Inducing developers to improve the quality of source code lexicon.* In 1990 Chikofsky and Cross [1990] defined the continuous reverse engineering, i.e., "reverse engineering, used with evolving software development technologies, will provide significant incremental enhancements to our productivity". Inspired by continuous reverse engineering, we conjecture that *continuous textual analysis* could be one of the solutions for inducing developers to improve the quality of source code lexicon. Preliminary study indicated that developers are induced to improve the source code lexicon if the software development environment provides information about the textual similarity between the source code under development and the related high-level artifacts [De Lucia et al, 2011].
- ***$O_2$**: Automatic setting on textual analysis process.* Approaches for solving the problem of assembling IR-based solutions for a given SE task and accompanying dataset are required. Our underlying assumption, which is supported by a large body of empirical research in the field, is that it is not possible to build a set of guidelines for assembling IR-based solutions for a given set of tasks as some of these solutions are likely to underperform on previously unseen datasets. Thus, automatic approaches need to be defined to find an optimal setting of the TA technique given a specific dataset.
- ***$O_3$**: Empirical studies.* Existing approaches based on textual retrieval have been evaluated off-line, i.e., they have been applied on ended software projects and the benefits have been evaluated considering some metrics. However, such approaches are intended to be used by software engineers. Thus, they need to be validated on-line, i.e., when used by real users during the development of software systems. Only in this way it is possible to measure the actual benefits of TA in supporting software engineering tasks.

## References

[Antoniol et al, 2002] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, Recovering traceability links between code and documentation, IEEE TSE, 28, 10, 970–983, 2002.

[Baeza-Yates and Ribeiro-Neto, 1999] Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley, 1999.

[Bavota et al, 2013a] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, A. De Lucia, An Empirical Study on the Developers Perception of Software Coupling, Proc. of ICSE, San Francisco, USA, 2013, 692-701.

[Bavota et al, 2013b] G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, A. De Lucia. Improving software modularization via automated analysis of latent topics and dependencies. TOSEM, 2013.

[Bavota et al, 2013c] G. Bavota, A. De Lucia, A. Marcus, R. Oliveto. Automating extract class refactoring: an improved method and its evaluation. EMSE, 2013.

[Bavota et al, 2013d] G. Bavota, A. De Lucia, A. Marcus, R. Oliveto. Using structural and semantic measures to improve software modularization. EMSE, 2013.

[Briand et al, 1998] L. C. Briand, J. W. Daly, and J. Wüst, A unified framework for cohesion measurement in object-oriented systems, EMSE, 3, 1, 65-117, 1998.

[Canfora and Cerulo, 2005] G. Canfora and L. Cerulo, Impact analysis by mining software and change request repositories, in Proc. of METRICS, Como, Italy, IEEE CS Press, 2005, 20–29.

[Chidamber and Kemerer, 1998] S. R. Chidamber, C. F. Kemerer. A metrics suite for object-oriented design. IEEE TSE, 20,6, 476-493, 1994.

[Chikofsky and Cross, 1990] E. J. Chikofsky, James H. Cross II: Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software 7(1): 13-17 (1990).

[Deerwester et al, 1990] S. Deerwester, S. T. Dumais, G. W. Furnas, T.K. Landauer, R. Harshman. Indexing by latent semantic analysis. JASIST, 41, 6, 391–407, 1990.

[De Lucia et al, 2011] A. De Lucia, M. Di Penta, R. Oliveto: Improving Source Code Lexicon via Traceability and Information Retrieval. IEEE TSE, 37,2, 205-227, 2011.

[Fowler, 1999] M. Fowler, Refactoring: improving the design of existing code. Addison-Wesley, 1999.

[Guerrouj et al., 2012] L. Guerrouj, P. Galinier, Y.-G. Guéhéneuc, G. Antoniol, M. Di Penta: TRIS: A fast and accurate identifiers splitting and expansion algorithm. Proc. of WCRE, Kingston, Canada, IEEE Press, 2012, 103-112.

[Marcus and Maletic, 2001] A. Marcus and J. I. Maletic, Identification of high-level concept clones in source code, in Proc. of ASE, San Diego, California, USA, IEEE CS Press, 2001, 107–114.

[Marcus et al, 2008] A. Marcus, D. Poshyvanyk, and R. Ferenc, Using the conceptual cohesion of classes for fault prediction in object-oriented systems, IEEE TSE, 34, 2, 287–300, 2008.

[Porter, 1980] Porter, M.F.: An algorithm for suffix stripping. Program, 14, 3, 130–137, 1980.

[Poshyvanyk et al, 2007] D. Poshyvanyk, Y. Gael-Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval, IEEE TSE, 33, 6, 420–432, 2007.

[Poshyvanyk et al, 2009] D. Poshyvanyk, A. Marcus, R. Ferenc, T. Gyimòthy. Using information retrieval based coupling measures for impact analysis, EMSE, 14, 1, 5–32, 2009.